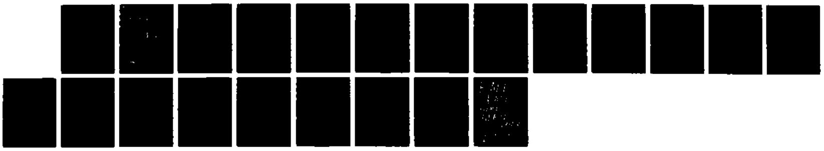
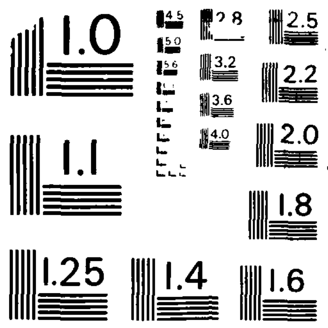
AD-A189 639 AN EFFICIENT PARALLEL ALGORITHM FOR THE SIMULATION OF THREE-DIMENSIONAL C..(U) INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERIN.. 1/1

UNCLASSIFIED G ERLEBACHER ET AL. JUL 87 ICASE-87-41 F/G 20/4 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6

1

NASA Contractor Report 178340

ICASE REPORT NO. 87-41

ICASE

AN EFFICIENT PARALLEL ALGORITHM FOR THE SIMULATION OF THREE-DIMENSIONAL COMPRESSIBLE TRANSITION ON A 20 PROCESSOR FLEX/32 MULTICOMPUTER

G. Erlebacher
S. H. Bokhari
M. Y. Hussaini

Contract No. NAS1-18107
July 1987

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

NASA
National Aeronautics and
Space Administration
Langley Research Center
Hampton, Virginia 23665

An Efficient Parallel Algorithm for the Simulation of Three-Dimensional Compressible Transition on a 20 processor Flex/32 multicomputer

G. Erlebacher
NASA Langley Research Center

Shahid H. Bokhari*
University of Engineering and Technology, Lahore, Pakistan

M.Y. Hussaini*
Institute for Computer Applications in Science and Engineering

The compressible, three-dimensional, time-dependent Navier Stokes equations are solved on a 20 processor Flex/32 computer. The code is a parallel implementation of an existing code operational on the Cray 2 at NASA Ames which performs direct simulations of the initial stages of the transition process of wall-bounded flow at supersonic Mach numbers. Spectral collocation in all three spatial directions (Fourier along the plate and Chebyshev normal to it) insures high accuracy of the flow variables. By hiding most of the parallelism in low level routines, the casual user is shielded from most of the non-standard coding constructs. Speedups of 13 out of a maximum of 16 are achieved on the largest computational grids.

* Research supported by the National Aeronautics and Space Administration under contract No. NAS1-18107 while resident at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23665.

I. Introduction

This paper describes the implementation and measured performance of a three-dimensional spectral algorithm on the Flex/32 shared memory multiprocessor. This algorithm was originally designed for, and run on, the Cray-2 as a conventional uniprocessor code to study the transition of compressible flow over a flat plate.

In view of the anticipated availability of multiprocessing on the current supercomputers (e.g. Cray-2 and ETA-10) and the scope for parallelism in the algorithm, this code was modified for parallel operation on the Flex/32. The Flex/32 was available and running at the time this work was started, while the ultimate target machines such as the ETA-10 and Cray-2 (with multiprocessing) had not been delivered.

The work reported in this paper has two significant aspects. First, a new spectral code has been developed to simulate the incipient stages of transition of supersonic, three-dimensional wall-bounded flows. Direct simulations of the initial stages of compressible transition will further our understanding of the basic mechanisms that ultimately result in a turbulent state. Current interest in aeronautics is centered around the development of high speed transports. In these regimes, very little is known about the transition process and how to control it. Nonetheless, the design of critical airframe components is affected by the extent of transition. The direct simulation of transition will ultimately lead to better transition models than those currently available. The cost of performing these simulations is unfortunately extremely high, even on today's most powerful supercomputers. Therefore it is important to investigate the possibility of speeding up the computation on multiprocessor computers. In this paper, we explain how the code is transformed to simultaneously execute on multiple processors with a high degree of parallelism. It is expected that the implementation of this algorithm on super multiprocessors like the Cray-2 or ETA-10 will effectively allow a practical number of parametric studies to be made in a reasonable time. Second, it is demonstrated that small multiprocessors based on cheap microcomputers can indeed be used as "work benches" for the development of parallel algorithms targeted for super multiprocessor computers.

After a description of the architecture of the Flex, the physical problem and its numerical discretization are explained. Design philosophy and the actual parallel strategy are then examined. As in all parallel algorithms, the issue of synchronization is important and must be carefully examined. A simple barrier that has a minimal impact on the original code has been developed and is explained. This is followed by numerical results, a comparison of the architecture of the Flex and two supercomputers, and a concluding section.

II. Architecture of the Flex/32 multiprocessor

The Flex/32 at NASA Langley Research Center has twenty processing elements based on the 32-bit National Semiconductor 32032 microprocessor [1]. Two processors are located on each "local" bus. The ten local busses are connected to a global bus. Access to shared global memory is through first the local and then the global bus. While the shared memory is accessible to all twenty processors, the local memory is private; a processor cannot access another processor's local memory.

The processors in the machine at Langley are numbered 1 through 20. Processor 1 and 2 run Unix and are used to develop and link programs and to load up and boot the remaining processors. Processors 3 through 20 run the MMOS concurrent operating system and parallel programs are normally run on some subset of these. At present, processors 1 and 11 reside on the same local bus, processors 2 and 12 on another, and so on up to processors 10 and 20 (Fig. 1).

In the current configuration, the two Unix nodes and processor 3 each have 4 megabytes (Mbytes) of private local memory, while the remaining 17 processors each have one Mbyte of local memory. Local memory can be expanded to a maximum of 4 Mbytes per processor. All the processors have access to $2\frac{1}{4}$ Mbyte of shared memory through the global bus.

Measurements of memory access times reveal that both local and global memories have access times of about $1.0\,\mu sec$. These are the minimum times required to fetch a floating point datum of four bytes. Time to access data from shared memory is affected only very slightly by contention on the global bus. The bandwidth of the global bus is approximately 5 times that of the local busses. Therefore, no degradation of performance for problems involving 5 or less CPU's is expected because data accessed from shared memory must still access the local bus to reach the processor. The 32032 microprocessor has a clock speed of $100\,nsec$, twice the speed of the common bus, and 10 times the speed of the local bus. Therefore, the CPU is idle 9 out of every 10 cycles during an operand fetch from local memory.

Time to fetch an operand from shared memory and multiply by a constant in local memory is strongly dependent on the type of variable addressing chosen by the C or Fortran compiler. Analysis of the produced machine code, or direct coding in assembler is necessary to give correct speed estimates for particular operations. This has not been done in this study. The same variability holds if the constants are in local memory. Practical tests reveal that manipulating operands in local memory is at about 5% faster than if they were stored in shared memory [2].

Experiments have also revealed that contention on the global bus has a negligible effect on memory access time. However, this may not be true on the supercomputer for which this algorithm is targeted. For this reason we have chosen to store a maximum number of constants, that are simultaneously accessed by more than one processor, in local memory.

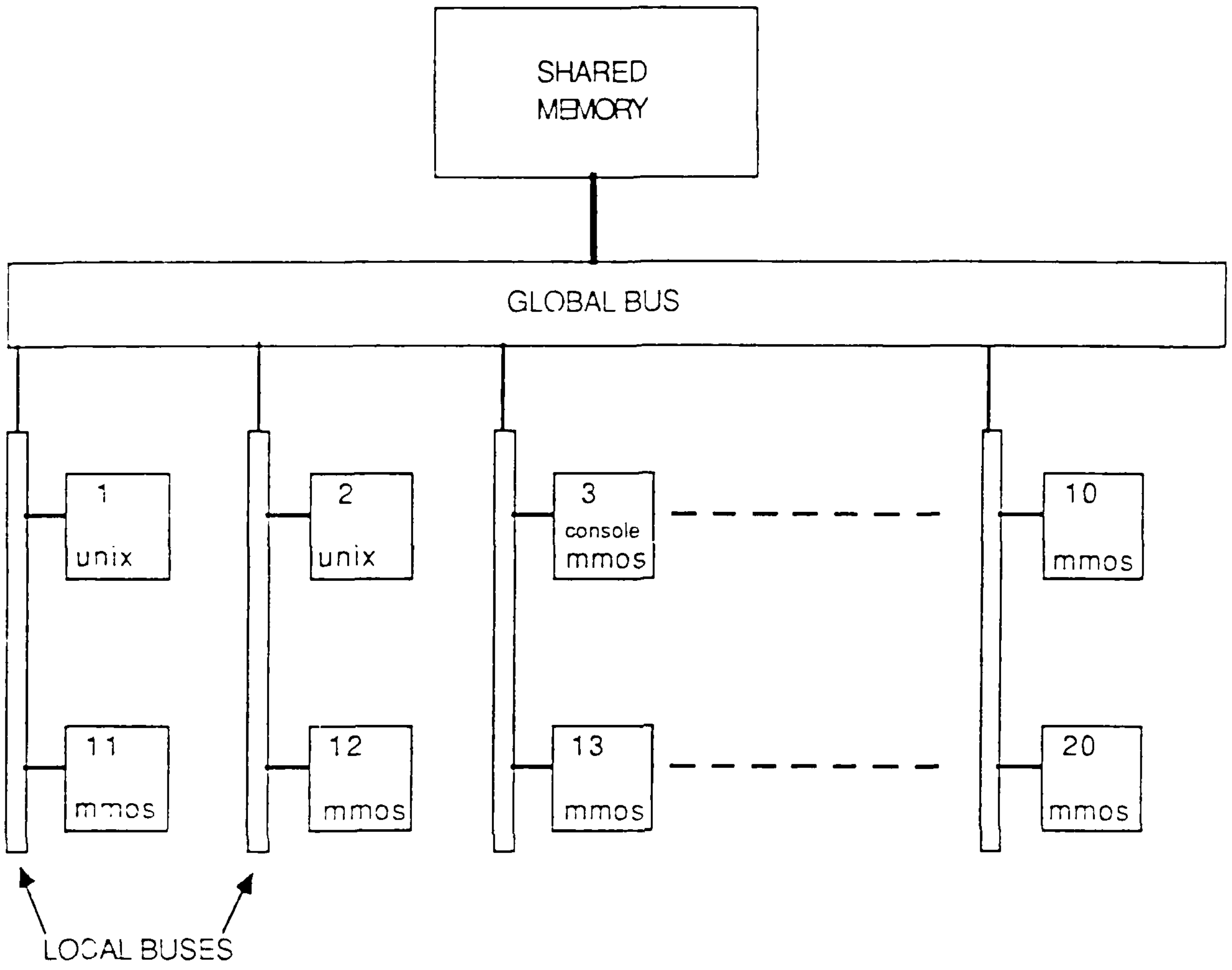


Fig. 1 Architecture of the Flex/32 Multicomputer

This is intended to avoid the queuing up of memory fetch requests which could slow down the transfer of information between processors and memory.

Therefore, for a low numbers of processors (< 5), the manner in which data is distributed throughout memory is not too important, but must be given careful consideration as the number is increased. For large number of processors, the Flex is a good *scale model* of realistic super multi-processor computers and design decisions adopted for the Flex are expected to be (at least partially) applicable to the larger machines.

III. Compressible Transition

Although, the prediction and understanding of transition of viscous flows has been studied experimentally, numerically and theoretically for the past 30 years, some of the fundamental mechanisms remain elusive. Most of the focus has centered on the analysis of incompressible flows because of their relative simplicity, both from the numerical and from the theoretical point of view. Prediction of transition is important in many areas of aerodynamics, particularly in the high speed regimes. Laminar, partially turbulent, or completely turbulent flow over a wing greatly influences the component's aerodynamical properties, its stress load, its weight, and ultimately the cost of production. Currently, the detailed analysis of the early stages of transition of supersonic flows over a flat plate is underway [3]. As the instabilities build up, nonlinearities set in, spatial length scales decrease as the velocities and temperature build up sharp gradients. As a consequence, more grid points are required to resolve the physics, along with more computer storage to store the data. This inevitably leads to subtantial amounts of processing time on today's largest supercomputers to solve even the simplest problems. Parallel computers potentially offer a way out of this dilemma by theoretically reducing the amount of wall clock time by a factor equal to the inverse of the total number of available processors. One approach is examined in this paper and preliminary results are given.

The full unsteady, three-dimensional, compressible Navier-Stokes equations are solved in the non-dimensional form

$$\frac{\partial \rho}{\partial t} + \nabla\cdot(\rho\vec{v}) \tag{1}$$

$$\frac{\partial(\rho\vec{v})}{\partial t} + \nabla\cdot(\rho\vec{v}\vec{v}) = -\nabla p + \nabla\cdot\vec{\vec{\sigma}} \tag{2}$$

$$\frac{\partial p}{\partial t} + \vec{v}\cdot\nabla\vec{v} = -\gamma p\nabla\cdot\vec{v} + \frac{1}{Re\,Pr\,M_\infty^2}\nabla\cdot(\kappa\nabla T) + (\gamma-1)\Phi \tag{3}$$

where ρ is the mass density, $\vec{v}$ is the velocity vector, p is the thermodynamic pressure, μ is the dynamic viscosity, h is the enthalpy, T is the absolute temperature, M_∞^2 is the free-stream Mach number, and κ is the thermal conductivity. The viscous stress $\vec{\vec{\sigma}}$ and the viscous dissipation Φ are respectively defined by

$$\vec{\vec{\sigma}} = -\tfrac{2}{3}\mu\nabla\cdot\vec{v} + \mu(\nabla\vec{v} + \nabla\vec{v}^T) \tag{4}$$

$$\phi = -\tfrac{2}{3}\mu(\nabla\cdot\vec{v})^2 + \frac{\mu}{2}(\nabla\vec{v} + \nabla\vec{v}^T)^2 \tag{5}$$

The Einstein summation convention applies to repeated indices and the effects of external body forces have been neglected for simplicity. Equations (1)-(3) must be supplemented with the equation of state for an ideal gas,

$$\gamma M_\infty^2 p = \rho R T, \tag{6}$$

where R is the ideal gas constant. Likewise, the dependence of the viscosity and thermal conductivity on the temperature must be provided (i.e., relationships of the form $\mu = \mu(T)$ and $\kappa = \kappa(T)$ are needed and these depend on the gas under study). Details are provided in [3].

Variables have been non-dimensionalized as follows: distance with respect to boundary-layer displacement thickness δ^*, velocity, temperature and density with respect to their free-stream values U_∞ , T_∞ , ρ_∞ and pressure with respect to the dynamic pressure $\rho_\infty U_\infty^2$. Viscosity and conductivity are scaled with respect to their free-stream values μ_∞ and κ_∞. With these definitions, the Reynolds number is

$$Re = \frac{\rho_\infty \delta^* U_\infty}{\mu_\infty}. \tag{7}$$

The Prandtl number

$$Pr = \frac{C_p \mu_\infty}{\kappa} \tag{8}$$

is assumed constant and equal to 0.72. C_p is the specific heat at constant pressure. The temperature dependent viscosity is given by the non-dimensional form of Sutherland's law

$$\mu(T) = \frac{(1 + 196.8 T_\infty)}{(T + \frac{196.8}{T_\infty})} T^{\frac{3}{2}}. \tag{9}$$

In these simulations of compressible transition, the growth of the mean boundary layer is suppressed through the addition of forcing terms to Eqs. (1)-(3). This is equivalent to studying the flow in a small region along the streamwise direction of the plate, in a neighborhood determined by the Reynolds number. Under this parallel assumption, the flow is assumed to be periodic in the two directions parallel to the plate.

IV. Spectral methods

Transition of viscous flows involves careful monitoring of a wide range of spatial length scales. The underlying mechanisms in transition involve the growth of instabilities which ultimately lead to turbulent breakdown [5]. In the absence of turbulence modeling, numerical algorithms must be extremely accurate and minimize the required computer memory storage for a given problem size. Although, the largest supercomputers can accommodate simulations on grids as large as 192 $\times$ 128 $\times$ 192, transition of laminar flow to a turbulent state has not yet been achieved numerically. Spectral algorithms have the property that for smooth flows, the discrete solution converges exponentially fast towards the exact solution. Therefore, on a fixed size grid, they will be more accurate than other numerical techniques above some critical number of points. More details are given in [6]. Put another way, for a given level of accuracy, spectral methods require less storage than alternative numerical strategies.

In return, derivative operators become full matrices and are more difficult to manipulate. Among the various types of spectral methods (Galerkin, tau and collocation), collocation is the most suitable for the treatment of the cubic nonlinearities that appear in the compressible momentum equations. Indeed, for cubic terms, convolution methods, as used in Galerkin methods applied to incompressible flows, are no longer suited for efficient transfer of information back and forth from physical to transformed space. Collocation methods expand the primitive variables into a linear combination of global basis functions. As explained in the previous section, periodicity is assumed in the streamwise and spanwise directions, and therefore the natural basis functions are Fourier functions. In the vertical directions, the flow variables are expanded in Chebyshev polynomials. In practice, the numerical algorithm is executed in physical space. The spectral representation only affects the computation of derivatives, and is transparent to the user.

The streamwise (x) and spanwise (y) directions (parallel to the plate) are respectively subdivided into n_x and n_y equally spaced cells, while the vertical direction (z) is discretized according to

$$z(\eta_k) = \frac{(a\eta_k + d)}{(b + c\eta_k)} \tag{10}$$

where

$$\eta_k = -\cos(\frac{\pi k}{n_z}) \qquad k = 1, \cdots, n_z \tag{11}$$

defines the Chebyshev collocation points. Equation (10) is a stretching of the collocation points to improve the viscous stability limit. A side effect of Chebyshev collocation is the severe $1/n_z^4$ dependence of the diffusive time limit.

Several long simulations on the Cray 2 have determined that almost all the work in the code is spent in the residual calculation, out of which over 90% of the CPU time is spent in routines that calculate the partial derivatives. Therefore, it is necessary to implement a highly efficient algorithm to evaluate derivatives in all three directions on the Flex/32 to obtain an overall good performance of the code.

In the context of spectral algorithms, derivatives are either calculated with Fast Fourier Transforms (FFT's) or with matrix multiplication (MM). A one-dimensional derivative calculation based on FFT's requires $5n_x \log_2 n_x + n_x$ floating point operations, while MM has an operation count of $2n_x^2$. For large n_x, the FFT is cheaper. In the interest of simplicity, we choose to implement our derivative routines with MM. Our program is written in such a way as to permit a switchover to FFT's with minimal disturbance to the code. Such a switchover to FFT's would reduce the overall running time of the code, without significantly impacting the efficiency of the intrinsic parallelism.

If D_x is the x derivative matrix, the derivative of a three-dimensional variable u(x,y,z) at node index (j,k,l) becomes

$$\left.\frac{df}{dx}\right|_{x_j, y_k, z_l} = \sum_{i=0}^{n_x - 1} D_{ij} f_{jkl} \qquad \left\{ \begin{array}{lcl} j & = & 0, n_x - 1 \\ k & = & 0, n_y - 1 \\ l & = & 0, n_z - 1 \end{array} \right. \tag{12}$$

For a Fourier expansion, the derivative matrix, F, (in the x direction) is defined as

$$\begin{array}{ll} F_{ij} = 0.5\ \alpha\ (-1)^{i-j} \cot(\frac{\pi}{n_x}(i-j)) & i \neq j \\ F_{ij} = 0.0 & i = j \end{array} \tag{13}$$

and a similar expression for the y derivative where the streamwise wavenumber α is replaced by β, the spanwise wavenumber. The discrete derivative matrix C in the normal direction, derived from an expansion in Chebyshev polynomials, is

$$\begin{aligned} C_{jk} &= \frac{c_j}{c_k}\frac{(-1)^{j+k}}{\eta_k - \eta_j} && j \neq k \\ C_{jj} &= -\frac{1+2(n_z-1)^2}{6} && j = 0, n_z - 1 \\ C_{jj} &= \frac{1}{2}\frac{c_j}{(\sin(\frac{j\pi}{n_z-1}))^2} && j \neq 0, n_z \end{aligned} \tag{14}$$

and

$$c_j = \begin{cases} 1 & j \neq 0, n_z - 1 \\ 2 & j = 1, \cdots n_z - 2 \end{cases} \tag{15}$$

In terms of the stretched variable z, the normal derivative becomes

$$\left.\frac{df}{dx}\right|_j = \sum_{i=1}^{n} C^*_{ij} f_j \tag{16}$$

where the new derivative matrix C^*_{ij} includes the contribution of the Jacobian of the coordinate transformation in the z direction. As a function of C_{ij},

$$C^*_{ij} = C_{ij} \left.\frac{\partial \eta}{\partial z}\right|_{\eta=\eta_j} . \tag{17}$$

A. Boundary and Initial Conditions

Periodic boundary-conditions are applied to all variables in the streamwise and spanwise directions. Normal to the plate, no slip conditions are applied to the velocities at the wall which is adiabatic. In the far-field, all the variables are frozen at their initial values.

The initial conditions consist of a triad of waves superimposed on a mean flow. Mean flow profiles are generated from the solution of the compressible boundary-layer equations. The perturbation waves are 2 and 3D Tollmien-Schlichting waves which are eigenfunctions of the linearized Navier-Stokes equations. These initial conditions, lead to a K-type secondary instability [7].

B. Discretization

To reduce the storage requirements on the Flex, a third order low-storage, explicit Runga-Kutta method is used for time discretization [4]. The primary advantage of this approach is that only arrays for the primitive variables and the residuals are needed. This is in contrast to a standard 3rd order Runga-Kutta method which consumes 3 arrays per variable. The numerical algorithm is fully explicit. However the efficiency of the algorithm is not adversely affected by an implicit algorithm in the vertical direction. This is because each processor receives the task of computing the implicit equations at a subset of the horizontal (x, y) points.

The time step is limited by both the convective (CFL_c) and the diffusion (CFL_d) time limits. In practice, a fixed time step is initially specified and the convective and diffusive Courant numbers are calculated at every time step. They are respectively defined by

$$CFL_c = 5.0\,\Delta t\, \max_{grid}\left(\frac{v_x}{\Delta x} + \frac{v_y}{\Delta y} + \frac{v_z}{\Delta z}\right) \tag{18}$$

and

$$CFL_d = 2.0\Delta t\, \max_{grid}\left[\mu\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2}\right)\right], \tag{19}$$

where v_x, v_y, v_z are the three component of velocity. If either Courant number exceeds unity, code execution is aborted.

V. Implementation

The compressible Navier-Stokes program was ported from the Cray 2 at Ames Research Center to the Flex/32 [8]. While the program is written in Fortran, it interfaces with modules written in C which contain all the pertinent initialization routines and the parallel constructs. If the program were rewritten today, the interface library would also be written in Fortran. However, when the Flex/32 first became available, the Fortran compiler was not fully debugged.

Sixteen three-dimensional arrays are required by the program. These include 5 arrays for the primitive variables $\rho, \rho\vec{v}, p$, 5 arrays for the residuals, and 6 temporary arrays. These provide working space for the derivative calculations, and allow a reduction in the number of derivative evaluations during the the residual calculation. Because derivatives are required in all three spatial directions, it is not possible to partition the primitive variable arrays across local memory without substantially increasing the flow of information between processors. Therefore, the cuboid of space on which computations are carried out is stored in shared memory. The storage required for the 16 arrays is $64 n_x n_y n_z$ which theoretically limits the total number of grid points to 64×32^2 if all the 3-D arrays are stored in common memory. However storage requirements of scalar arrays and of the operating system limit the maximum grid size to $32 \times 32 \times 16$. For problems of practical interest, very large grids are necessary (for direct simulations). For examples, grids of 128^3 and beyond are necessary to reach the initial stages of breakdown in incompressible flows [4]. This trend is expected to hold for compressible flows. In the linear regime, grids of $16 \times 16 \times 64$, adequate for Mach numbers below 4.5, barely fits into the memory of the Flex.

The computational grid is non-adaptive, so the the components of the derivative matrices remain fixed in time and are calculated once when the code is initiated. The matrices are stored in local memory. This is important because derivative matrices stored in global memory would lead to memory contention. For example, assume that processors 3 and 4 are both calculating an z-derivative, processor 3 along the line (x_1, y_1, z) and processor 4 along (x_2, y_2, z). If both units are operating at exactly the same rate with no communication overhead, they will each access the identical matrix elements at the same time. This will reduce the effective speed of the computation because a given memory location cannot be accessed instantaneously by several processors. Furthermore, if there are 10 or more processors computing z-derivatives, the increased flow of data on the global bus will further decrease the efficiency of the algorithm.

The constant matrix used for computing derivatives is stored in shared memory. When the parallel computation begins, each processor copies this matrix into local memory. Each processor then independently computes the subsection of the cuboid for which it is responsible during the calculation of x, y and z partial derivatives. This is possible because the size of the computational domain and the total number of active processors is known to each processor. For example, when using 4 processors on a $16 \times 32 \times 64$ grid, each processor is responsible for a grid of size $16 \times 32 \times 64/4$ when computing x-derivatives, and of size $(16/4) \times 32 \times 16$ when y and z derivatives are required.

Over 90% of the processing is spent inside the partial derivative routines. Even though a speed increase can be sought through optimization of the derivative evaluation on one or multiple processors, the purpose of this paper is to demonstrate efficiency, not speed. Any optimization will affect all processors equally, and therefore the efficiency of the parallel algorithm remains invariant[†]. Grid sizes in each direction are currently limited to integer multiples of the number of active processors for simplicity.

At program initiation, the derivative matrices are computed in common memory and copies are made to each processor's local memory. In the current version of the program, each processor has stored in its local memory a copy of the complete program. In a `maxprocs` processor simulation, the processes are logically numbered from 0 to `maxprocs`-1. The logical process number (proc_nb) is passed to each process by the master process which initiates the program. The slab thickness in the z direction is calculated according to the prescription

```
delprocz = zsize / maxprocs;
lzfrom = delprocs * proc_nb;
lzto   = lzfrom   + delprocz;
```

where `zsize` is the number of points in the z direction (corresponding to n_z in the previous section), and `delprocz` is the slab thickness. Similar relations are defined in the two other

[†]This statement is true as long as the communication and synchronization times are negligible compared to the amount of work per processor.

directions. The slab seen by a specific process is delimited by the indices `lzfrom` and `lzto` along z. Node numbering in the code ranges from 0 to `zsize-1` in the z direction, and similarly in the x and y directions which respectively have `xsize` and `ysize` nodes. Extra slab variables related to grid partitioning are defined for the Fortran portion of the program whose loops range from 1 to the number of nodes in any given direction. In this paper, to avoid confusion, it is assumed that loop index limits are the same in both languages.

The bulk of the work is performed during the residual calculation. Because spectral collocation is used in all three directions, and is a global method, derivative calculations can easily be isolated from the rest of the code. They are computed through the use of

```
    call partial(i,u,dux)
```

where `i` is 1, 2 or 3 depending on whether the derivative is in the x, y or z directions, u is the three-dimensional dependent variable, and dux is the derivative returned by `partial`. In its turn, `partial` calls one of three lower level routines that actually compute the derivatives on the appropriate slabs. Apart from computing derivatives, the code combines them to build up the actual residuals. For illustration purposes, consider the typical fragment on a serial machine:

```
    call partial(1,u,dux)
    call partial(2,v,dvy)
    call partial(3,w,dwz)

    do 20 k=1,zsize
       do 10 ij=1,xsize*ysize
          rhs(ij,1,k)=rhs(ij,1,k)
             -dux(ij,1,k)-dvy(ij,1,k)
             -dwz(ij,1,k)
 10    continue
 20 continue
```

which calculates the divergence of the velocity, and subtracts it from the right hand side (`rhs`). This term appears in the continuity equation. Loop 10 ranges over the entire $x-y$ plane. In the parallel version of the code, loop 20 is replaced by

```
    do 20 k=lzfrom,lzto.
```

Each processor calculates its own value of `lzfrom` and `lzto` as described above.

Common blocks in Fortran are identified with structures in C. They are both defined as either local or shared variables through specific Concurrent C or Fortran constructs [9,10].

After transferring the global derivative matrices to local arrays, the C code calls the Fortran program with the two arguments `lfromz` and `ltoz`. In this manner, the changes imposed on the original source code are minimized. Apart from modifications to various loop index ranges, declarations of variables in the various subroutines must be explicitly declared local or global.

To further increase the potential efficiency of the algorithm, many one dimensional arrays and constants are stored in local memory. Among these, are the Mach number and related physical parameters, and the grid characteristics, including the complete node distribution in the vertical direction.

Finally the issue of synchronization must be considered. All the processes initiate the derivative computation simultaneously and synchronize themselves before entering either another derivative calculation or a `DO` loop. This issue is addressed next.

VI. Synchronization

As in most non-trivial parallel programs, a key issue is the synchronization of processes and/or processors. Proper synchronization is essential for the generation of non-corrupted results. For example, consider what happens with the following Fortran code fragment which contains no synchronization constructs:

```
      call partial(2,u,duy)
      do 10 k = lfromz, ltoz
         do 20 ij = 1, xsize*ysize
            u(ij,1,k) = u(ij,1,k)
                    + duy(ij,1,k)
   20    continue
   10 continue
```

which calculates the y derivative (duy) of the 3-D array u, and adds it to u. Each processor first calculates its share of the x-derivative over the subset `xsize/nb_procs,ysize,zsize` of the grid. The index of loop 10 partitions the domain into slabs perpendicular to the z axis. If processor 1 finishes its share of the derivative calculation before the other processors, processor 1 could access elements of duy along the x axis with the range of k defined by loop 10 before their update in `partial` by the other processors. To avoid potential data corruption, it is necessary to synchronize the processors before computation of partial derivatives and before most loop constructs.

The implementation of the synchronization was done so that it does not intrude unnecessarily in the actual program code. Synchronization operations should be encapsulated in subroutines so that there is no possibility of accidental modification to crucial operations.

All the user sees is a call to the C routine `lsynch()` which acts as a barrier which is released when all the processors have reached it.

The Flex provides us with `lock/unlck` and `when` (condition) constructs [1]. These can be called from *C* routines which can be linked into Fortran programs. The `when` statement causes the executing process to suspend until the specified condition is true. This statement alone cannot provide the sort of synchronization that is required. The most straightforward barrier construct is the single lock strategy

```
 1   lsynch()
 2   {
 3      lock(1,"Gsteps"); Gsteps++;
                    unlck(1,"x");
 4      when (Gsteps == maxprocs) ;
 5      if (Gsteps == maxprocs)
 6         Gsteps = 0;
 7   }
```

where on entry, `Gsteps=0`. The problem with this approach is that when the last processor reaches line 3, `Gsteps` receives its final increment and the first processor can exit the barrier (line 4) and potentially reset `Gsteps` to 0 before all the processors have reached line 5. If this occurs, the processors become desynchronized and the program hangs. The remedy is one version of the double lock mechanism [11]:

```
 1   lsynch() {
 2      lock(1,"Gsteps"); Gsteps++;
               unlck(1,"Gsteps");
 3      when (Gsteps == maxprocs);
 4      lock(1,"Gh");
 5      if (Gh == maxprocs)
 6         Gh = 0;
 7      unlck(1,"Gh");
 8      lock(1,"Gh"); Gh++;
             unlck(1,"Gh");
 9      when (Gh == maxprocs);
10      lock(1,"Gsteps");
11      if (Gsteps == maxprocs)
12         Gsteps = 0;
13      unlck(1,"Gsteps");
14   }
```

where `Gsteps` is initialized as before, and `Gh` is initially set equal to `maxprocs`. The major difference between the single and double lock barrier is the use of two `when` statements (lines 3 and 9). When the last processor performs the final update of Gsteps (line 2), all processors fall through the barrier at line 3. The first one through resets Gh to 0. The second barrier at line 9 stops all the processors until the last processor reaches line 8 and Gh is equal to maxprocs. The first processor through the barrier (line 9) resets Gsteps to maxprocs. There is no longer the possibility of desynchronization because Gh was reset *before* reaching the second barrier. Only if there is an insufficient amount of work between successive calls to lsynch could a synchronization problem possibly occur. However, the large grids preclude this. One remedy to insure a foolproof program is to use different locks every time lsynch is called. More general parallel languages (e.g. the Force [12]) have built in barrier constructs (in Fortran) which could have been used. Unfortunately, the language was not available at the time this work was in progress, and moreover, the Fortran compiler was not reliable.

VII. Measured performance on Flex

The full time-dependent, compressible, 3-D Navier-Stokes equations are solved on the Flex/32 using 1, 2, 4, 8 and 16 processors. Each time step requires three computations of the residual. Since the residual calculation is by far the most computationally expensive, the time required to initialize the code, to perform I/O, and to compute various diagnostics is neglected. This is valid because the equations are integrated for many time steps without I/O or diagnostics. In the extreme limit, the code only consists of residual calculations.

The efficiency of parallel algorithms (from the engineer's point of view) ultimately boils down to the amount of CPU time saved as the number of processors (maxprocs) is increased. In this spirit, three measures of efficiency are computed, based on measured CPU time, as a function of the total number of processors, P. The total CPU time (T_P) is the time required to execute one full residual calculation. Speedup (S_P) is measured as the ratio of the time per residual on P processors over the time spent calculating one residual on a single processor. The maximum speedup is, of course, simply the number of processors. Efficiency (E_P), yet another method of gauging algorithm efficiency is defined by

$$E_P = \frac{S_P}{P}. \tag{20}$$

A maximum efficiency of 1 is equivalent to the statement that every processor is busy computing all the time. Results from several runs on computational grids ranging from $16 \times 16 \times 16$ to a maximum size of $32 \times 32 \times 16$ are presented in Table I. Measured times correspond to one complete residual calculation (in CPU seconds). The ordering of the grids (from top to bottom) is in decreasing order of the total number of floating point operations for 3 derivative calculations (in x, y and z). This number is proportional to

$$N_{tot} = n_x n_y n_z (n_x + n_y + n_z). \tag{21}$$

since the number of derivative calculations in each spatial direction is the same. The work performed by each processor during this set of 3 matrix multiplies is N_{tot}/P. This is

Performance					
Grid	$N_{tot} \times 2^{-14}$	P	T_P	S_P	E_P
$128 \times 16 \times 8$	12.7	8	365	7.55	94.3
		4	705	3.91	97.6
		2	1386	1.99	99.4
		1	2757	1.00	100.0
$8 \times 64 \times 32$	9.0	8	269	7.52	94.0
		4	510	3.87	96.8
		2	1003	1.97	98.5
		1	1977	1.00	100.0
$64 \times 16 \times 16$	8.0	16	138	13.01	81.3
		8	242	7.41	92.6
		4	466	3.84	95.9
		2	916	1.95	97.7
		1	1786	1.00	100.0
$32 \times 32 \times 16$	6.7	16	118	12.82	80.1
		8	202	7.45	93.2
		4	388	3.89	97.3
		2	759	1.99	99.3
		1	1511	1.00	100.0
$32 \times 16 \times 16$	2.7	16	62	9.98	62.4
		8	92	6.72	84.0
		4	168	3.67	91.7
		2	321	1.92	96.0
		1	616	1.00	100.0
$16 \times 16 \times 16$	1.0	16	37	6.54	40.9
		8	43	5.60	70.0
		4	71	3.44	86.0
		2	127	1.92	95.8
		1	244	1.00	100.0

Table 1: Performance data of one residual calculation

corroborated by the results in Table I which indicate that the time required to compute one residual increases with N_{tot} for a fixed number of processors. As expected, for a constant N_{tot}, the speedup increases with P. The highest efficiency with 8 active processors occurs when N_{tot} is the largest. In this case, the work per processor is maximum, while the time spent during communication and synchronization processes has not changed much as a function of N_{tot}. Speedups of over 7.5 out of possible maximum of 8 have been achieved on the $128 \times 16 \times 8$ grid. With 16 processors active, maximum speedups of 13 were measured. However, it is clear that this number could increase with larger grids. Unfortunately, memory limitations prevent this hypothesis from being tested.

VIII. Implementation on the Cray-2 and the ETA-10

The Flex/32 offers a testbed to try a parallel algorithm on a physical problem of paramount interest. Indeed, the wide range of spatial length scales demands very fine resolution, while the non-stationary character and strongly non-linear character of the flow leads to the requirement of huge computer resources. Today, two supercomputers potentially lend themselves to the practical study of such problems. They both provide parallel processing capability, but the distribution of resources is different on the two systems. These computers are the Cray 2 and the ETA-10. A brief description is given of both computers, together with the advantages and disadvantages relative to the proposed algorithm.

A. Cray 2

The Cray 2 has 4 processors, each with a peak performance of 480 Mflops (with all functional units enabled). It comes with either 64 MWords (fast) or 256 MWords (slow) of common memory [13]. In addition to common memory, a local cache of 16 kWords is available to all processors. The two types of memory on this machine are very similar to the local/global memory of the Flex and one would expect to port the program without too much difficulty to the Cray 2. However, on the Cray 2, the amount of local memory is very limited and thus only some of the local variables used in the Flex implementation can be stored in it. Furthermore, at the time of writing, it is not clear how this memory may be accessed by Fortran programs and how much of it is available. In the worst case, we would store variables in shared memory. Performance degradation due to memory conflicts has yet to be ascertained.

B. Eta-10

An ETA-10 is essentially 8 Cyber 205's hooked up in parallel. Each processor has a peak speed of approximately 600 Mflops in 64 bit arithmetic (when using linked triads) and has 16 Mwords of local memory. This is supplemented by 256 Mwords of global memory.

The ETA-10 and the Flex/32 are thus also very similar in their distribution of memory resources. However, usage of the common memory is not transparent as in the case of the Flex/32 in the sense that variables cannot be declared in local or common memory. All variables are stored in local memory unless explicitly transferred to global storage. The rationale behind this approach is related to the very high Mflop rate relative to the communication speed between local and common memory. According to the researchers at ETA systems, a rule of thumb is that a good algorithm should perform 5 floating point operations per word transferred one way from common memory. When porting our implementation to the ETA 10, we will explicitly move slabs of the compu'atational domain between shared and local memory as the computation proceeds. These transfers will inevitably generate an overhead which remains to be determined.

IX. Conclusions

The three-dimensional, time-dependent, compressible Navier-Stokes equations have successfully been implemented on the 20 processor Flex/32 at Langley Research Center. A spectral algorithm provides the accuracy required for the study of transitional flows in the supersonic regime. On the largest grids, speedups of 13 on 16 processors were observed. The speedup might increase further as the work load on each processor increases. Unfortunately, the limited amount of common memory precluded tests with grids larger than $32 \times 32 \times 16$.

The implementation described in this paper can be ported to the Cray 2 or ETA 10 with some minor modifications. This could lead to enormous reductions in the computer time required to complete a specific simulation. In turn, this could be the first step towards achieving the capability of processing large numbers of direct simulations of 3D compressible stability problems.

X. Acknowledgements

We would like to thank Tom Crockett for his help and advice while the Flex was still an alpha-site at Langley.

References

[1] Flexible Computer, *Multicomputing Multitasking Operating System Reference Manual*. Publication No. 030-0004-002, 2nd Edition.

[2] Crockett, T., Private Communication, 1987.

[3] G. Erlebacher and M.Y. Hussaini, "Stability and Transition in Supersonic Boundary Layers," AIAA paper No. AIAA-87-1416, 1987.

[4] G. Erlebacher, "Incipient Transition Phenomena in Compressible Flows over a Flat Plate," ICASE Report No. 86-39, 1986.

[5] L. S. Kovasznay, H. Komoda and B. R. Vasudeva, Proc. of the 1962 Heat Transfer and Fluid Mech. Inst, pp. 1-26, Stanford University Press, 1962.

[6] M. Y. Hussaini, "Spectral Methods in Fluid Dynamics," *Ann. Rev. Fluid Mech.*, Vol. 19, pp. 339-367, 1987.

[7] T. Herbert, "Three-Dimensional Phenomena in the Transitional Flat-Plate Boundary-Layer," AIAA 23rd Aerospace Sciences Meeting, Paper No. AIAA-85-0489, 1985.

[8] Bokhari, S., Erlebacher, G. and Hussaini, M. Y., "Three-Dimensional Compressible Transition on a 20 processor Flex/32 Minicomputer," Proceedings of the Sixth IMACS International Symposium on Computer Methods for Partial Differential Equations held at Lehigh University, PA, U.S.A., Ed. by R. Vichnevetsky and R.S. Stepleman, 444-451, 1987.

[9] Flexible Computer, *ConCurrent FORTRAN Reference Manual*. Publication No. 030-0004-002, 2nd Edition.

[10] Flexible Computer, *ConCurrent C Reference Manual*. Publication No. 030-0003-002, 1st Edition.

[11] T. S. Axelrod, "Effects of Synchronization Barriers on Multiprocessor Performance," *Parallel Computing*, Vol. 3, 129-140, 1986.

[12] H. Jordan, "The FORCE on the FLEX: Global Parallelism and Portability," ICASE Report No. 86-54, 1986.

[13] Cray Inc., *Cray-2 Computer System Functional Description* HR-2000, 1985.

Standard Bibliographic Page

1. Report No. NASA CR-178340 ICASE Report No. 87-41	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle AN EFFICIENT PARALLEL ALGORITHM FOR THE SIMULATION OF THREE-DIMENSIONAL COMPRESSIBLE TRANSITION ON A 20 PROCESSOR FLEX/32 MULTICOMPUTER		5. Report Date July 1987
		6. Performing Organization Code
7. Author(s) G. Erlebacher, S. H. Bokhari, and M. Y. Hussaini		8. Performing Organization Report No. 87-41
9. Performing Organization Name and Address Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225		10. Work Unit No. 505-90-21-01
		11. Contract or Grant No. NAS1-18107
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, D.C. 20546		13. Type of Report and Period Covered Contractor Report
		14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor: J. C. South — Submitted to AIAA J.

Final Report

16. Abstract

The compressible, three-dimensional, time-dependent Navier-Stokes equations are solved on a 20 processor Flex/32 computer. The code is a parallel implementation of an existing code operational on the Cray 2 at NASA Ames which performs direct simulations of the initial stages of the transition process of wall-bounded flow at supersonic Mach numbers. Spectral collocation in all three spatial directions (Fourier along the plate and Chebyshev normal to it) insures high accuracy of the flow variables. By hiding most of the parallelism in low level routines, the casual user is shielded from most of the non-standard coding constructs. Speedups of 13 out of a maximum of 16 are achieved on the largest computational grids.

17. Key Words (Suggested by Authors(s)) compressible Navier-Stokes equations, parallel computations	18. Distribution Statement 34 - Fluid Mechanics and Heat Transfer; 64 - Numerical Analysis; Unclassified - unlimited

19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of Pages 19	22. Price [illegible]